View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Universidad de San Buenaventura, sede Bogotá: Editorial Bonaventuriana

EVALUATION OF GCC OPTIMIZATION PARAMETERS

Rodrigo D. Escobar
The University of Texas at San Antonio
rodavesco@gmail.com

Alekya R. Angula
The University of Texas at San Antonio
a.alekyareddy@gmail.com

Mark Corsi
The University of Texas at San Antonio
mark@corsi.us

(Tipo de Artículo: **Investigación**. Recibido el 10/11/2012. Aprobado el 23/12/2012)

ABSTRACT
Compile-time optimization of code can result in significant performance gains. The amount of these gains varies widely depending upon the code being optimized, the hardware being compiled for, the specific performance increase attempted (e.g. speed, throughput, memory utilization, etc.) and the used compiler. We used the latest version of the SPEC CPU 2006 benchmark suite to help gain an understanding of possible performance improvements using GCC (GNU Compiler Collection) options focusing mainly on speed gains made possible by tuning the compiler with the standard compiler optimization levels as well as a specific compiler option for the hardware processor. We compared the best standardized tuning options obtained for a core i7 processor, to the same relative options used on a Pentium4 to determine whether the GNU project has improved its performance tuning capabilities for specific hardware over time.

Keywords
Compiler optimization, Machine Learning, Compiler Heuristics, Programming Languages, Processors.

EVALUACIÓN DE PARÁMETROS DE OPTIMIZACIÓN GCC

RESUMEN
La optimización en el tiempo de compilación del código puede resultar en ganancias de rendimiento significativas. La cantidad de dichas ganancias varía ampliamente dependiendo de código a ser optimizado, el hardware para el que se compila, el aumento que se pretende en el desempeño (e.g. velocidad, rendimiento, utilización de la memoria, etc.) y el compilador utilizado. Se ha utilizado la versión más reciente de la suite de benchmarks SPEC CPU 2006 para ayudar a adquirir la comprensión de las mejoras posibles en el desempeño utilizando las opciones GCC (GNU Compiler Collection) que se concentran principalmente en las ganancias de velocidad fueron posibles ajustando el compilador con los niveles de optimización del compilador estándar así como una opción de compilador específica para el procesador de hardware. Se compararon las opciones más estandarizadas de ajuste obtenidas para un procesador core i7, para las mismas opciones relativas utilizadas sobre un Pentium4 para determinar si el proyecto GNU ha mejorado sus capacidades de ajuste de desempeño para el hardware especifico en el tiempo.

Palabras clave
Optimización de compilador, Aprendizaje automático, Heurística de compiladores, Lenguajes de programación, Procesadores.

ÉVALUATION DE PARAMÈTRES D'OPTIMISATION GCC

Résumé
L'optimisation du temps de compilation du code peut résulter dans profits significatifs de rendement. La quantité de tels profits change largement selon le code à être optimisé, le hardware pour lequel on compile, l'augmentation prétendue dans le rendement (e.g. vitesse, rendement, utilisation de la mémoire, etc.) et le compilateur utilisée. On a utilisé la version le plus récent de la suite de benchmark SPEC CPU 2006 pour aider à la compréhension des améliorations possibles sur le rendement en utilisant les options GCC (GNU Compiler Collection) qui se centrent essentiellement sur les profits de vitesse qu'ont été possibles en ajustant le compilateur avec les niveaux d'optimisation du compilateur standard, de même que une option de compilateur spécifique pour le processeur de hardware. On a comparé des options les plus standardisés d'ajustage obtenus pour un processeur core i7, pour les mêmes options relatives utilisés sur un Pentium4 pour déterminer si le projet GNU a amélioré ses capacités d'ajustage de rendement pour le hardware spécifique dans le temps.

Mots-clés
Optimisation de compilateur, Apprentissage automatique, heuristique pour compilateurs, langages de programmation, processeurs.

1. INTRODUCCIÓN

Compiling and optimizing software for a specific machine is more of an art than a science. Performance tuning is dependent upon so many factors that there is cutting edge work being performed on implementing machine learning as a methodology for determining the best compile time options [1] Due to this complexity, compilers - specifically the GNU Compiler Collection (GCC) have developed standardized compile time optimization switches (6 levels) along with many other flags which allow the user to control a lot of characteristics of the code produced by the compiler [2], [3], [4]. Among all these compiling options, GCC also have standardized compile time options for specific processor classes (e.g. Intel i7 core, Pentium 4, and others), which can be set using the mtune parameter.

Besides, the Standard Performance Evaluation Corporation (SPEC) has developed a well-known and widely used suite of integer and floating point CPU-intensive *benchmarks* for the purpose of testing and comparing hardware platforms [5]. The last version was released in 2006 and it is called SPEC CPU2006. SPEC CPU2006 consists of 12 integer and 7 floating point compute-intensive workloads called CINT2006 and CFP2006 respectively, which are provided as source code and may be compiled with different options. Thus, they can be used as a test bed to compare the performance of different computer architectures, as well as the efficacy of different compiler optimization options [2]. The CINT2006 suite measures the compute-intensive integer performance, while the CFP2006 suite measures the compute-intensive floating point performance [6].

GCC is arguably the most utilized, publicly available compiler in use today. It is provided as the default compiler on most Linux systems and it is a highly developed library which has been in production since the 80's [7]. Because it has been in production for such a long time, we could use it in combination with the SPEC CPU2006 Benchmark Suites and two vastly different hardware platforms to determine whether the de facto standard compiler had improved its performance-based tuning for specific hardware over the last decade.

The remainder of this paper is organized as follows: The next section includes the motivation. Section 3 presents a brief introduction to compilers and compiler optimizations. Section 4 describes the experimental setup and methodology. Our approach, results, and additional statistics of the optimizations are discussed in Section 5. Section 6 discusses future work, and finally Section 7 concludes.

2. MOTIVATION

A large variety of programs are executed every day on different processors available in the market. Most of these programs have been subject to a compilation process before their execution. Compilation processes are complex and involve lot of areas of computer science. In this paper, we aim to assess the performance of the GCC compiler when its different optimization parameters are set. Each optimization level includes over twenty optimizations options. Therefore, a detailed analysis of how each optimization technique affects the results is out of the scope of this paper. Instead, we try to get a general idea about how the GCC compiler has improved its capabilities to apply optimization techniques to specific processors. Compilers' configuration parameters that have been designed to generate optimized code for a particular hardware, such as the mtune parameter provided in the GCC compiler, are of particular interest. The motivation behind it is to see whether or not compilers are getting better over time at optimizing for specific architectures. Although machine learning may one day become the best way of optimizing the installation or compilation of software [1], it currently does not appear to be a solution that is ready for prime time production use. In the mean time, compilers will need to become even more efficient at utilizing the underlying system architecture through self-optimization. In this paper we examine options, also called switches, that are already present in GCC and that could easily be automated to be 'turned on' at compile time after examining the hardware it is running on. Moreover, we discuss whether work has been ongoing over the last decade into making optimizations for specific hardware stronger.

3. COMPILERS' OPTIMIZATION TECHNIQUES

Compilers are basically computer programs that translate a program written in a specific language to a program written in another language [8]. The source program is usually a program that has been written in a high level programming language, whereas the result of the translation procedure is usually a program ready to be executed by a processor (i.e. machine language code). A compiler must stick to two fundamental principles:

- A compiler must preserve the meaning of the source program.
- A compiler must improve the source program.

Preserving the meaning of the source program means that every time the translated program is executed, it produces exactly the same output that the source program would produce when supplied with the same input data. On the other hand, improvement of the source program can refer to several different characteristics of the resulting code, such as portability, size, energy consumption, or time of execution, among others.

The structure of a compiler is usually divided into three sections or phases, namely: Frontend, optimizer and backend. A three phase compiler structure is presented in Figure 1. Each phase usually includes several sub-phases.

3.1. Frontend

The goal of the frontend phase of a compiler is to transform the source code into some sort of intermediate representation (IR). An intermediate representation is a machine independent representation of the source program. The frontend phase is considered to be composed of the following phases:

- Lexical analysis: The text of the source program is divided in words or tokens and every word is categorized as a generic symbol in the programming language, e.g. a variable name, an identifier, a type of variable, etc.
- Syntax analysis: This phase takes the list of tokens produced in the lexical analysis pass and arranges them in a structure called parse tree that reflects the structure of the program.
- Type checking: This phase analyzes whether the program violates certain requirements such as declaring a variable more than once, assigning a boolean value to a string variable, etc.
- Intermediate representation generation: The main goal of this phase is to create a machine independent representation of the program. It can take different forms, such as a tree, a graph, or code.

3.2. Optimization

The intermediate representation of a program is transformed to produce an optimized version of it. Usually, each optimization technique is applied independently, which means that the intermediate representation of the program may be passed several times through the optimizer. Compilers which determine how to transform the code to run faster or consume fewer resources are known as optimized compilers. The goal of optimized compilers is to perform one or more safe and profitable transformations on the intermediate representation, preserving the results and the meaning of the program [9], [6]. Current advanced processors are dependent on the compilers to design the object code for the optimal performance. The GCC compiler consists of an intermediate language which is transformed by an independent representation of the program [9].

3.3. Backend

The goal of the backend phase is to take the intermediate representation of a program and produce machine code. This section is composed of the following phases:

- Register allocation: In this pass, symbolic variable names used in the intermediate code are assigned a register in the target machine code.
- Machine code generation: This is the phase that actually produces assembly code for a specific machine architecture.
- Assembly and linking: The assembly-language code generated in the previous pass is translated into a binary representation. Also, addresses of variables, functions, etc., are determined.

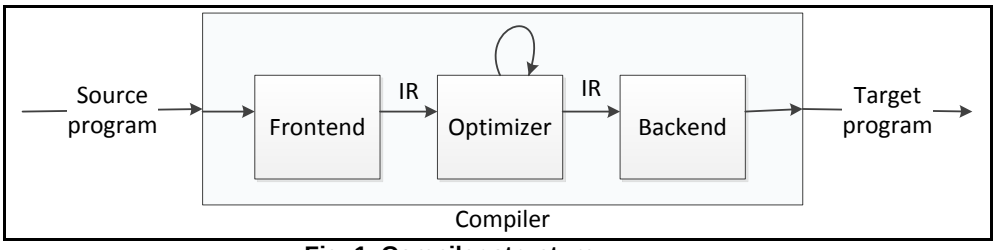


Fig. 1. Compiler structure

It is important to clarify that, although the term optimization implies that an optimal solution is found for a particular problem, compilers in practice face problems that cannot be solved optimally. Therefore, compilers aim to improve the source code in a safe and profitable way [8]. For a compiler to apply any optimization technique to a program, it must do the following three things [10]:

- Decide what section of a program to optimize and the particular transformation to apply.
- Verify that the transformation will preserve the semantics of the source program.
- Transform the program

Compiler optimization techniques can be machine dependent or machine independent. Machine dependent optimizations depend on the specific hardware in which the translated program is intended to be executed. On the other hand, machine independent optimizations do not depend on a particular computing system or a type of implementation [11].

4. EXPERIMENTAL SETUP

For our experiments we used two machines, namely Machines A and C which are described in Table 1. In order to perform comparisons, we compiled and ran the SPEC CPU2006 benchmarks in machine A –the machine with the most modern hardware, using GCC's generic optimization levels O0, O1, O2 and O3 [12]. GCC's optimization levels include different optimization techniques that are performed during compilation time. Table 2 presents the set of options applied for each GCC's optimization level [13].

For each one of the optimization levels we computed the geometric mean among all the workload speedups and, according to it, we identified the best optimization level. Unsurprisingly, as shown in table 3, the best results were obtained from the standardized

optimization level O3. Later, we appended to the best optimization level found (i.e. O3), the mtune compiler parameter which tries to optimize the code produced by GCC for a specific processor (i.e. with the mtune parameter, GCC tries to perform machine dependent optimizations) [14]. Therefore, for Machine A we appended the parameter mtune=corei7 to the SPEC's Optimize configuration flags, and then we executed the resulting binary files. From now on, we identify these binary files as the mtuned binary files or mtuned programs and correspond to programs resulting from a compilation process that uses the optimization parameters O3 and mtune according to each target machine (i.e. mtune=i7 for Machine A and mtune=Pentium4 for Machine C).

Although the use of the mtune configuration parameter when compiling any source code doesn't decrease the portability of the programs, it aims to increase the performance of the program execution when it is run in the processor specified with the mtune parameter [14]. Consequently, since we are running the mtuned binary files in machines with the same processors that were established for the mtune parameter, we expected to achieve a more significant improvement for our mtuned programs than the improvement we found in the previous execution of the benchmarks with only the O0, O1, O2 and O3 optimization levels.

Since Machine A's processor is clearly much more modern and powerful than Machine C's processor, comparing the results obtained when compiling using an optimization level in Machine A directly against the results obtained when compiling using the same optimization level in Machine C does not make sense. Instead, our purpose is to compare the relative differences between compiling in Machine A against the difference obtained when compiling in Machine C. Thus, once we ran the mtuned benchmarks in Machine A, we compiled and ran the benchmarks in Machine C using the O0 optimization level and later using the best compilation level found for Machine A (i.e. level O3) with the mtune parameter set to match the specific processor of Machine C (i.e. Pentium 4).

Table 1. Systems' specifications

	Machine A	**Machine C**
CPU Name	Intel core i7 – 2630QM	Intel P4
CPU MHz	2000	2400
FPU	Floating point unit	Integrated
CPUs enabled	4 cores, 1 chip, 4 cores/chip, 8 threads/core	1 core, 1 chip, 1 core/chip
Primary cache	32 KB I + 32 KB D on chip per core	64 KB I + 64 KB D on chip per chip
Secondary cache	256 KB I+D on chip per core	512 KB I+D on chip per chip
Level 3 cache	8 MB I+D on chip per chip	None
Other cache	None	None
Memory	4 GB	4 GB
Disk subsystem	SATA	IDE
Operating system	Linux (Ubuntu 11.10)	Linux (CentOS 5)
Compiler	GCC 4.6.1 (gcc, g++, gfortran)	GCC 4.6.1 (gcc, g++, gfortran)

The results are presented in Table 4. After running the benchmarks in both machines, we compared the differences between the obtained O0 and mtuned (i.e. O3 + mtune) results of each machine separately. We expected the differences to be smaller for Machine C, than for machine A, and consequently show that GCC has become a better architecture specific optimizer over the last decade.

Table 2. GCC's optimization options

GCC'S OPTIMIZATION LEVEL	**INCLUDED OPTIMIZATION TECHNIQUES**
O0	No options enabled.
O1	Combination of increments or decrements of addresses with memory accesses, reduction of scheduling dependencies, dead code elimination, avoidance of popping the arguments to each function call as soon as that function returns, reordering of instructions to exploit instruction slots available after delayed branch instructions, dead store elimination, guess branch probabilities, transformation of conditional jumps into branch-less equivalents, discovery of functions that are pure or constant, perform interprocedural profile propagation, merge identical constants, loop header copying on trees, constant/copy propagation, redundancy elimination, range propagation, expression simplification, hoisting of loads from conditional pointers on trees, scalar replacement of aggregates and temporary expression replacement during SSA.
O2	All the options of O1, and also performs: contained branch redirections, alignment of the start of functions, alignment of branch targets, alignment of loops, alignment of labels, cross-jumping transformation, common subexpression elimination, convert calls to virtual functions to direct

GCC'S OPTIMIZATION LEVEL	INCLUDED OPTIMIZATION TECHNIQUES
	calls, function inlining, interprocedural scalar replacement of aggregates, removal of unused parameters, replacement of parameters passed by reference by parameters passed by value, enable peephole optimizations, reassignment of register numbers in move instructions and as operands of other simple, reordering of basic blocks in the compiled function in order to reduce number of taken branches and improve code locality, reordering of functions in the object file in order to improve code locality, schedule instructions across basic blocks, allow speculative motion of non-load instructions, reordering of instructions to eliminate execution stalls and conversion of simple initializations in a switch to initializations from a scalar array, value range propagation.
O3	All the options of O2 and also performs: movement of branches with loop invariant conditions out of the loop, predictive commoning optimization, loop vectorization on trees and function cloning to make interprocedural constant propagation stronger.

5. RESULTS

In order to keep the variables to a minimum for meaningful comparison, we focused only on the Integer suite of the SPEC CPU2006 bechmarks. Our initial assumption was that GCC would become stronger at optimizing for later model processors using the mtune parameter, accepting as a given that during ten years of development (the amount of time lapse from the oldest P4 processor to the newest i7 quad core production) tuning skills would have been improved.

Our assumptions were predominantly wrong with several exceptions. But the exceptions alone proved to be interesting to our initial assumptions.

In computer architecture terminology, the term Speedup is defined as [15]:

$$Speedup = \frac{Runtime\ without\ enhancement}{Runtime\ using\ enhancement}$$

Most of the tests performed within a narrow range of median performance speedup. In other words, when we normalized each tests performance with the oldest processor's baseline we found that the later model processor showed a fairly consistent speedup across the board without varying greatly from the base normalized mean. Tuning increased the performance nearly across the board within a discrete range. Tables 3 and 4, show the obtained results for machines A and C respectively. Shown ratios are compared to the SPEC's 2006 benchmarks reference machine. For simplicity, only the geometric mean values after executing all the different benchmarks are presented. However, more detailed data are presented in Figures 2 and 3, which show the obtained results for machines C and A respectively. Shown ratios are compared to machine C. Data corresponding to machine C are presented first, because Machine A speedup ratios shown in Figure 3 are relative to Machine C's O0 results.

Unfortunately for our assumptions, the mtune optimization coupled with the best performance optimization (O3) on average produced a very small performance increase when compared to generic O3 optimization by itself. We had initially theorized that this gap, the gap between performance of the best generic optimization (O3) and the best generic optimization combined with specific processor optimization (mtune), would widen on the later model processors. With one major exception, this did not occur. Consequently, it seems from the results we obtained, that GCC's developments are not very focused on per processor tuning and consequently, in general, users should not expect a significant improvement by configuring GCC to produce optimized code for a specific hardware. Furthermore, sometimes such code may be less efficient than a code that has been compiled for generic hardware.

Table 3. Optimization on Core I7 using SPEC CPU 2006 Benchmarks

	Core i7					
	O0		O3		O3 -mtune=corei7	
	Seconds	Ratio	Seconds	Ratio	Seconds	Ratio
Geometric Mean	892.73	11.33	436.97	23.13	428.57	23.57

Table 4. Optimization on Pentium 4 using SPEC CPU 2006 Benchmarks

	Pentium 4					
	O0		O3		O3 –mtune = pentium4	
	Seconds	Ratio	Seconds	Ratio	Seconds	Ratio
Geometric Mean	2912.71	3.48	1858.88	5.44	1976.18	5.12

Detailed examination of the instrumentation readout provided insights into the most prevalent phenomena (i.e. the overall increase in processor throughput over the last ten years). Although this is of course, highly predictable; what was surprising is the most predominant reason for it as shown by detailed instrumentation readout. The processors have similar overall clock speeds. In fact, the Machine A's processor has a slightly slower clock speed than Machine C. However predictably, the later processors have multiple cores, larger caches and faster IO (bus speed and hard drive technology). Surprisingly, the most dominant factor in overall speed gains appears to be major page faults.

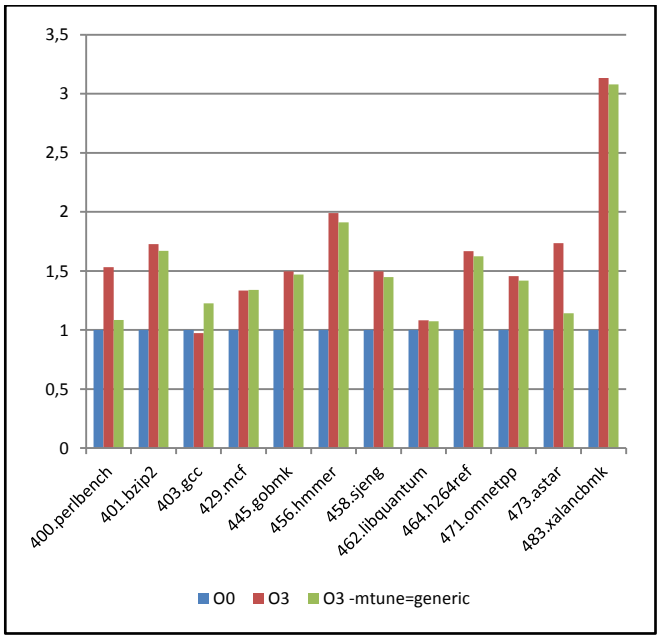


Fig. 2. Normalized Ratios and Runtime on Pentium4 using SPEC CPU 2006 Benchmarks

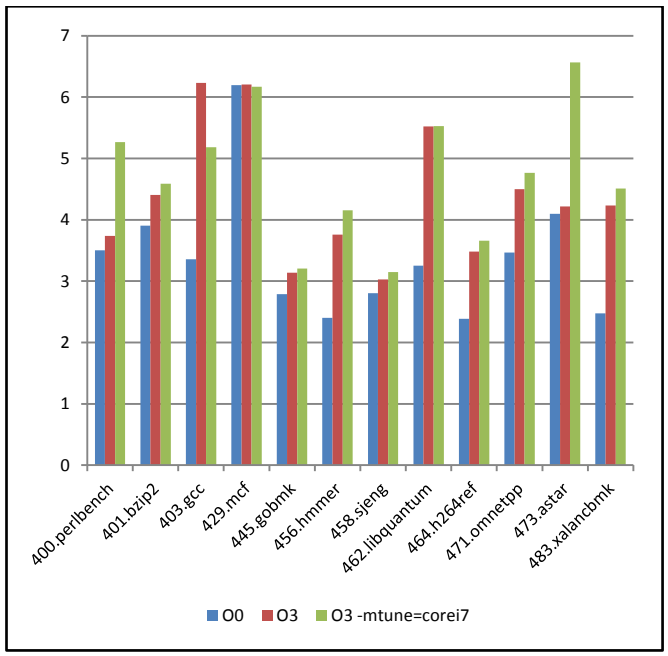


Fig. 3. Normalized Ratios and Runtime on Core I7 using SPEC CPU 2006 Benchmarks

There are two types of page faults: minor and major. From figures 4 and 5, it is clear that minor page faults increase with the later processors and major page faults decrease. Exploring this phenomenon a little further, we can determine this makes sense. The oldest processor is a single core processor with a single L2 Translation Look-aside Buffer (TLB). Minor page faults will only occur on this machine when another process makes use of a particular page referenced in this table. However on the multicore machines - which currently do not have a shared last level TLB minor page faults can occur whenever a thread is spawned from the same process on a different core. Hence, the more cores not sharing the TLB, the more minor page faults. However, this is inconsequential as an overall indicator of speed since the virtual pages are still in memory; they are simply not marked in the memory management unit as loaded. This 'soft page' fault is easily fixed and does not create a large performance hit.

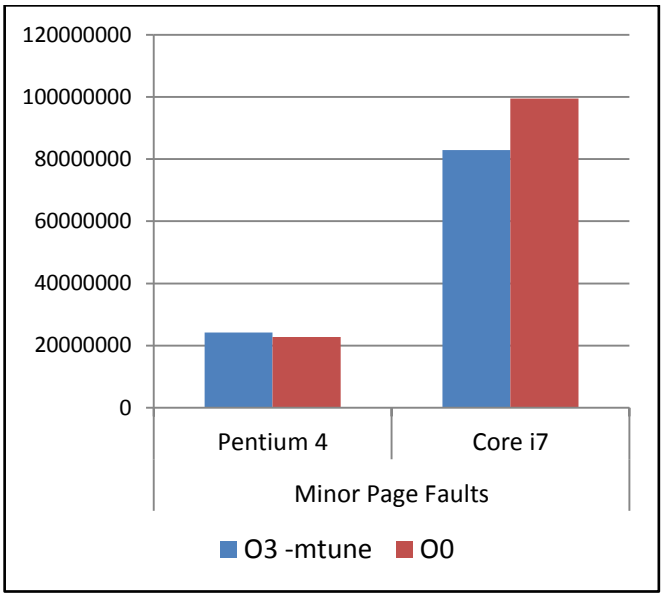


Fig. 4. Comparing minor Page faults between both machines

From Figure 4 we can see that the minor page faults are more for Machine A when compared to Machine C. O3-mtune has less number of minor faults when compared to the -O3 optimization. Nevertheless, Figure 5 shows that major faults are less for Machine A than for Machine C. It can also be noticed that after tuning, the major page faults are reduced in Machine C. In both processors the tuning helps in reducing the number of page faults.

Statistics related to major page faults are presented in Figure 5. Looking at the normalized numbers, we can see that decreased major page faults are easily attributable as a large part of the later systems' overall performance increase. It can be argued that this one statistic alone is responsible for between 46 - 91 percent of the overall system performance gains. We say, 'it can be argued' because even though these numbers are normalized, we do not have exact indicators as to the performance cost of a page fault vs

the performance cost of another slow down e.g. cache misses. However, we do know that page faults are probably the single biggest performance slow down in a system due to the IO time required to retrieve a page from disk.

Bearing in mind the foundations presented in the previously exposed points, we now focus on analyzing the results presented in Figure 3. Figure 3 presents the speedup ratios relative to those of Machine C, which were present in Figure 2. The interesting tests were: 403.gcc, 462.libquantum, 473.astar, 429.mcf. These 4 out of 12 benchmarks displayed anomalies outside the normal range of speedups shown by the majority. The fourth one displays an across the board speedup nearly double that of the normal range. And more interesting still, are the other three which we believe indicate a fundamental difference in the way GCC tunes for later processors.

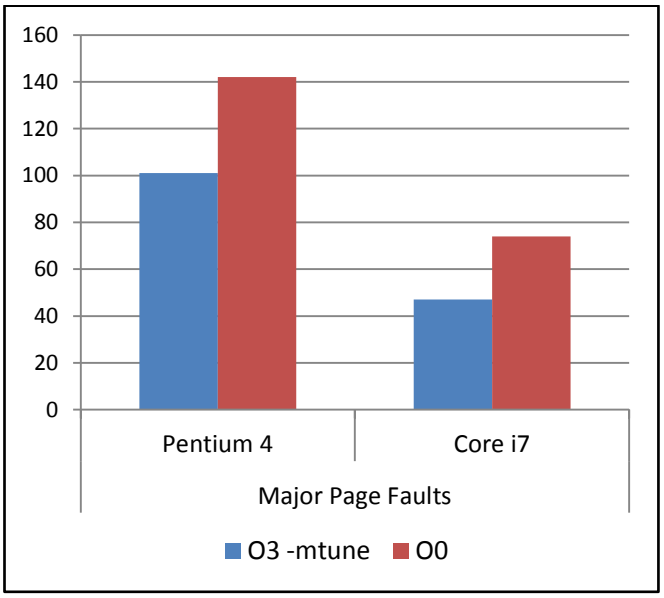


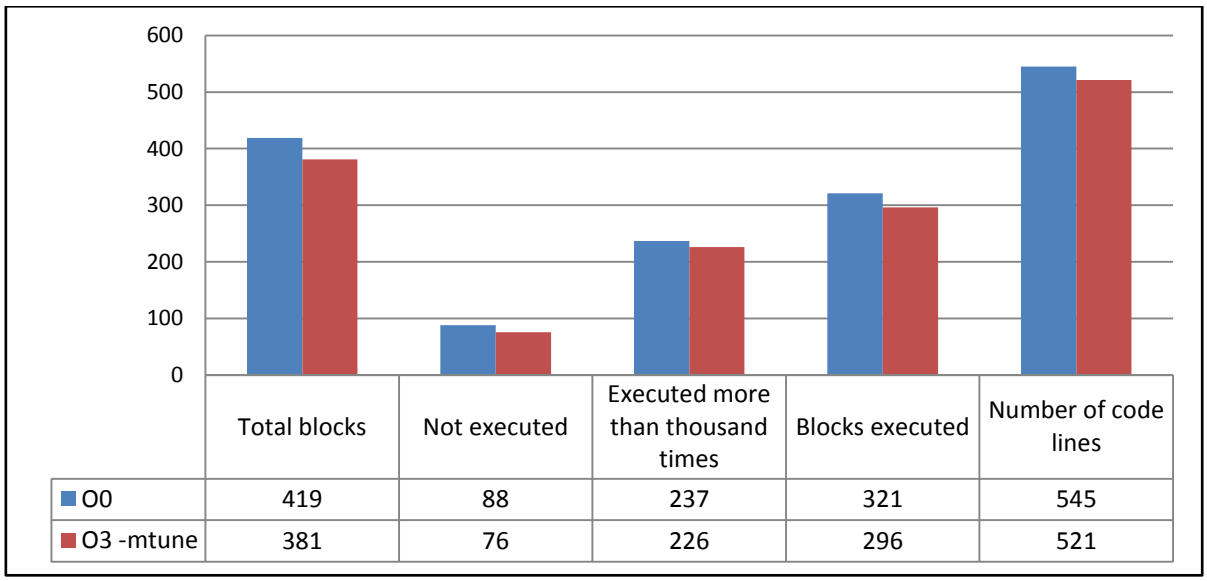
Fig. 5. Comparing major Page faults between both machines

First, we will discuss the fourth test, which does not indicate a better tuning strategy on the part of GCC developers, but does show the largest across the board performance increase (over 6x speedup for Machine A when compared to Machine C; these speedups occurred on all three SPEC INT tuning tests –O0 baseline, -O3 and –O3 tuned for the processor). The test was the 429.mcf test which revolves around combinatorial analysis. The nature of the test, predictable branching and a high level of locality – both spatial and temporal, allows it to maximize the greatest improvements of the newer processor. This is also shown in Figure 6, the test has less basic blocks and a much higher percentage of them are accessed over 1000x when compared to the other tests indicating that stronger caching mechanisms will produce much better results.

The other three tests that exhibit anomalies are of greater interest. The first, 403.GCC shows a huge normalized improvement with the O3 optimization, but a marked decrease in performance when using the O3 tuning coupled with the specific processor tuning. Figure 7 shows basic blocks counting for this benchmark. Although we were unable to determine a reason for the decreased performance for mtuned later processors, the huge increase in the O3 performance appears to be at least partially due to loop unrolling (note the decrease in all block numbers for O3 tuning in Figure 7).

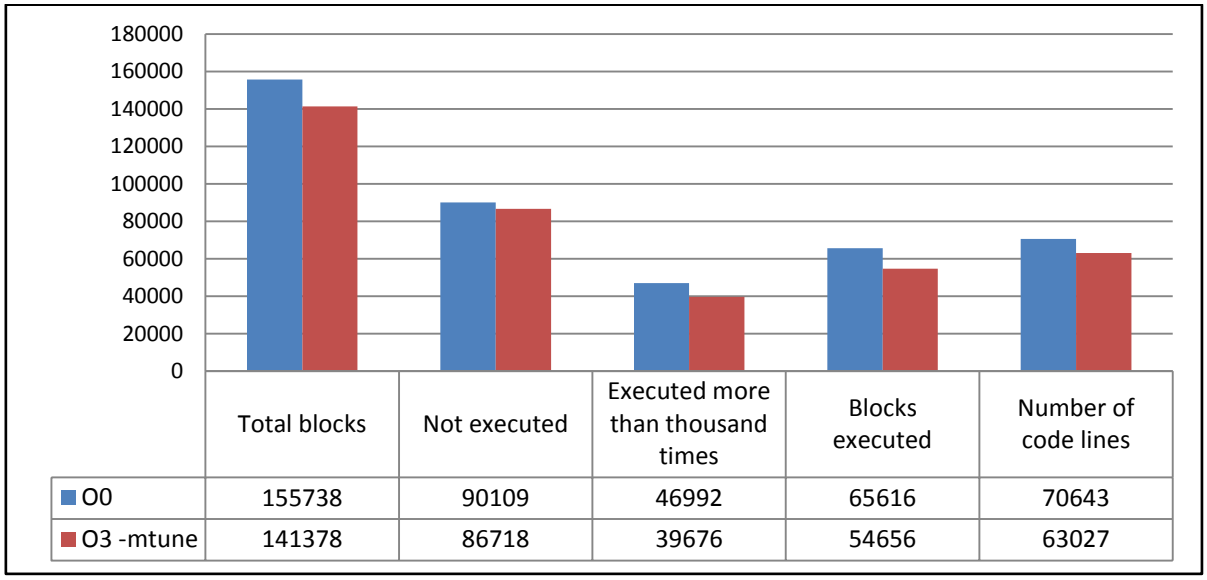
The second, 462.libquantum which simulates a quantum computer, also shows a relatively huge O3 optimization gain on the newer processors, but this time the specific processor tuning is more in line with the rest of the tests which means it is either about the same as the O3 alone, or slightly better. However, in this test, loop unrolling does not appear to be as much of a factor. We mention the anomaly simply because it is apparent in the raw data, but were unable to draw any firm conclusions based upon the data at hand.

And finally, the third test and by far the most interesting for our initial hypothesis, is the 473.astar test which revolves around path-finding algorithms. This was the only one test where our hypothesis proved correct as shown in Figure 3. Specific processor tuning (mtune) for GCC showed big performance increases relative to baseline and O3 testing for the dual core processor and huge increases for the i7. Path finding algorithms revolve around recursive coding which is highly thread-able. Each of the later processors has an increasing amount of hardware threads when compared to Machine C. The basic blocks for this test are not modified greatly during the O3 optimization stage so loop unrolling does not produce a sizable performance gain, however the mtune tests show slight gains in the dual core which is capable of spawning 4 simultaneous hardware threads and huge gains when placed on the highly threaded quad core processor.



	Total blocks	Not executed	Executed more than thousand times	Blocks executed	Number of code lines
O0	419	88	237	321	545
O3 -mtune	381	76	226	296	521

Fig. 6. Basic blocks using 429.mcf Benchmark



	Total blocks	Not executed	Executed more than thousand times	Blocks executed	Number of code lines
O0	155738	90109	46992	65616	70643
O3 -mtune	141378	86718	39676	54656	63027

Fig. 7. Basic blocks using 403.GCC Benchmark

6. FUTURE WORK

In order to firm up our conclusion, the benchmarks should be run against later model computers containing processors with different hardware threading capabilities, ideally two (one of each) with 16 hardware threads, two with 8 hardware threads and two with 4 threads. The number of cores and cache sizes should remain the same for each pair in order to eliminate other possible variables. The tests should then be normalized to the original P4 (2 thread) processor run. If our conclusions are correct, we should see normal improvements in O3 level optimization, but ever increasing improvements in the mtune optimization speed.

7. CONCLUSIONS

It appears GCC developers have done very little optimization development for specific processors. We believe this is due to the large number of possible processors in the market and the fact that tuning for each individual processor would be an arduous process at best. Further testing on an array of the latest, highly threaded processors would be necessary to concretely declare this as a truth.

REFERENCIAS

[1] M. Stephenson, S. Amarasinghe, M. Martin and U.-M. O'Reilly, "Meta optimization: improving compiler heuristics with machine learning," in *PLDI '03 Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, New York, 2003.

[2] Standard Performance Evaluation Corporation, "SPEC's Benchmarks and Published Results". Online [September. 2011].

[3] Standard Performance Evaluation Corporation, "SPEC CPU2006 Benchmark Descriptions". Online [September. 2011].

[4] K. Hoste and L. Eeckhout, "Cole: compiler optimization level exploration" in *CGO '08 Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, 2008.

[5] Standard Performance Evaluation Corporation, "SPEC CPU2006" Online [September. 2011].

[6] Wind River Systems, "Advanced compiler optimization techniques" April 2002. Online [December. 2012].

[7] D. Edelsohn, W. Gellerich, M. Hagog, D. Naishlos, M. Namolaru, E. Pasch, H. Penner, U. Weigand and A. Zaks, "Contributions to the GNU Compiler Collection" *IBM Systems Journal,* vol. 44, no. 2, p. 2005, 259-278.

[8] L. Torczon and K. Cooper, Engineering a Compiler, San Francisco, CA: Morgan Kaufmann Publishers Inc., 2007.

[9] W. v. Hagen, The Definitive Guide to GCC, Second Edition ed., Berkeley, CA: Apress, 2006.

[10] D. F. Bacon, S. L. Graham and O. J. Sharp, "Compiler transformations for high-performance computing" *ACM Computing Surveys (CSUR),* vol. 26, no. 4, pp. 345-420, 1994.

[11] P. B. Schneck, "A survey of compiler optimization techniques" in *ACM '73 Proceedings of the ACM annual conference*, 1973.

[12] T. Jones, "Optimization in GCC" January 2005. Online [December. 2012].

[13] The GNU Compiler Collection Team, "Options That Control Optimization" Online [October. 2011].

[14] The GNU Compiler Collection Team, "i386 and x86-64 Options – Using the GNU Compiler Collection" Online [October. 2011].

[15] J. L. Hennessy and D. A. Patterson, Computer Architecture: A Quantitative Approach 4 Ed, San Francisco, CA: Morgan Kaufmann Publishers Inc., 2006.